Final Project

ESSignments group, Bocconi Univeristy January 8, 2021

Implementation of Metropolis Algorithm

Below you can find the code we created to implement the Metropolis Algorithm with non-informative prior.

Listing 1: Python code – Metropolis Algorithm with Non-Informative Prior

```
1
2  import numpy as np
3  import scipy.stats as stats
4  import matplotlib.pyplot as plt
5  from datetime import datetime
6  from statsmodels.discrete.discrete_model import Probit
7  plt.style.use('seaborn')
8
9  startTime = datetime.now()
10 np.random.seed(250)
11
12 ###   ###   ###   ###   ###   ###
13
14 # SET UP THE MODEL
15
16 # Set the dimension of your problem, with N as the number of observations ↩
       and
17 # d as the number of parameters
18
19 n = 400
20 d = 4
21
22 # Generate a sequence of random numbers from a Normal distribution to then↩
       be able to fill in the matrix X
23
24 x = np.random.randn(n,d-1)
25
26 X = np.ones((n,d))
27 X[:,1:] = x
28
29 # Make a guess on the true value of theta
30
```

```
31 true_theta = np.array([0.5,0.2,0.3,-0.2]).T
32
33 # By making use of the probit link, compute the probabilities p (that is, ↩
       the mean) of the Bernoulli Multivariate model
34
35 p = stats.norm.cdf(X.dot(true_theta))
36
37 # Then again, simulate the outcomes by making use of the probabilities we ↩
       have.
38 # We used here a random draw from a binomial with parameter n = 1, to get ↩
       a
39 # random draw from a Bernoulli distribution
40
41 y = np.random.binomial(1, p, size=None)
42
43 # Initialize the theta
44
45 theta = np.zeros(d).T
46
47 ###   ###   ###   ###   ###
48
49 # DEFINITION OF THE VARIANCE OF THE PROPOSAL
50
51 # Proposal (Multivariate Normal centered at the current update with
52 # variance matrix given by the inverse of Fisher Information)
53
54 V = np.linalg.inv(-Probit(y,X).hessian(theta))
55
56 ###   ###   ###   ###
57
58 # DEFINITION OF THE TARGET FUNCTION
59
60 def target(theta):
61     return np.exp(Probit(y,X).loglike(theta))
62
63 ###   ###   ###   ###
64
65 # RUN THE MODEL
66
67 N_sim = 10000
68 burn_in = 5000
69 naccept = 0
70 theta_chain = np.zeros((N_sim,d))
71
72 # Iterations
73
74 for i in range(N_sim):
```

```
75      V = np.linalg.inv(-Probit(y,X).hessian(theta))
76      theta_p = np.random.multivariate_normal(theta,V)
77      rho = min(1, target(theta_p)/target(theta))
78      u = np.random.uniform()
79      if u < rho:
80          naccept += 1
81          theta = theta_p
82      theta_chain[i,:] = theta.reshape((1,d))
```

Then, we decided to include an informative prior in the Metropolis Algorithm. More specifically, we used a multivariate Normal centered at the current draw of theta with covariance matrix given by diagonal matrix of 10's. Here is the code that illustrates the changes we brought about in order to introduce the informative prior. As you can see, we added the covariance matrix and then changed the target by adding the evaluation of the pdf of the multivariate normal at the current draw.

Listing 2: Python code – Metropolis Algorithm with Informative Prior

```
1  # DEFINITION OF THE TARGET FUNCTION
2
3  Q_0 = np.eye(d)*10
4
5  def target(theta):
6      return stats.multivariate_normal.pdf(theta,theta,Q_0)*np.exp(Probit(y,↩
           X).loglike(theta))
```

It is important to note that we chose a Multivariate Normal prior because it is commonly used in the literature. It is also common to choose values of the diagonal covariance matrix that are large enough, to allow the prior of the parameter not to be drastically informative. Eventually, it is possible to see that the outcomes of the non-informative version are exactly the same as the ones provided by the informative algorithm. This is not surprising: putting an extremely large variance on an informative prior can give the same results as using a non-informative prior, since we allow the values to fluctuate in an extremely wide range. Finally, it is important to note that the multivariate normal distribution would be the conjugate prior of a Multivariate Probit Model.

Implementation of Auxiliary Gibbs Sampler

We started by running the algorithm with a non-informative prior, with the simulated coefficients being drawn from a multivariate Gaussian distribution with mean defined in line 76 and variance defined in line 60.

Listing 3: Python code – Auxiliary Gibbs Sampler Algorithm with Non Informative Prior

```
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt
from datetime import datetime
startTime = datetime.now()

np.random.seed(250)

###   ###   ###   ###   ###   ###

# SET UP THE MODEL

# Set the dimension of your problem, with N as the number of observations and
# d as the number of parameters

n = 400
d = 4

# Generate a sequence of random numbers from a Normal distribution to then be
# able to fill in the matrix X

x = np.random.randn(n,d-1)

X = np.ones((n,d))
X[:,1:] = x

# Make a guess on the true value of theta

true_theta = np.array([0.5,0.2,0.3,-0.2]).reshape((d,1))

# By making use of the probit link, compute the probabilities p (that is, the mean) of the Bernoulli Multivariate model

p = stats.norm.cdf(X.dot(true_theta))

# Then again, simulate the outcomes by making use of the probabilities we have.
# We used here a random draw from a binomial with parameter n = 1, to get a random draw from a Bernoulli distribution

y = np.random.binomial(1, p, size=None)


###   ###   ###   ###   ###   ###
```

```
42
43 # INITIALIZING THE PARAMETERS
44
45 theta_0 = np.zeros((d,1))
46
47 theta = theta_0
48 z = np.zeros((n,1))
49
50 N_sim = 10000
51 burn_in = 5000
52
53 theta_chain = np.zeros((N_sim,d))
54
55
56 ###   ###   ###   ###   ###   ###
57
58 # GIBBS SAMPLING ALGORITHM
59
60 V = np.linalg.inv(X.T.dot(X))
61
62 for t in range(1,N_sim):
63
64     # Update the mean of Z
65
66     mu_z = X.dot(theta)
67
68     # Draw latent variable Z from its full conditional
69
70     z[y == 0] = stats.truncnorm.rvs(a = -np.inf, b = -mu_z[y == 0], loc = ↩
          mu_z[y == 0], scale=1)
71
72     z[y == 1] = stats.truncnorm.rvs(a = -mu_z[y == 1], b = np.inf, loc=↩
          mu_z[y == 1], scale=1)
73
74     # Compute posterior mean of theta
75
76     M = np.linalg.inv(X.T.dot(X)).dot(X.T.dot(z))
77
78     # Draw variable theta from its full conditional
79
80     theta = np.random.multivariate_normal(M.ravel(),V).reshape((d,1))
81
82     theta_chain[t,:] = theta.T
83
84 # Get the posterior mean of theta
85
86 post_theta = theta_chain[burn_in:,].mean(axis=0)
```

Below you can find the code we created to implement the Auxiliary Gibbs Sampler by Albert and Chib. As you can see in line 41, we used the informative prior provided in the slides: the Normal with mean specified as in line 37 and variance specified in lines 20 and 21. In this paper, we chose not to rewrite the parts of the code that were the same as the non-informative algorithm. Nevertheless, in the zip file it is possible to encounter the complete version of the script.

Listing 4: Python code – Auxiliary Gibbs Sampler Algorithm with Informative Prior

```
1  ###   ###   ###   ###   ###   ###
2
3  # INITIALIZING THE PARAMETERS
4
5  theta_0 = np.zeros((d,1))
6  Q_0 = np.eye(d)*10
7
8  theta = theta_0
9  z = np.zeros((n,1))
10
11 N_sim = 10000
12 burn_in = 5000
13
14 theta_chain = np.zeros((N_sim,d))
15
16 ###   ###   ###   ###   ###   ###
17
18 # GIBBS SAMPLING ALGORITHM
19
20 prec_0 = np.linalg.inv(Q_0)
21 V = np.linalg.inv(prec_0 + X.T.dot(X))
22
23 for t in range(1,N_sim):
24
25     # Update the mean of Z
26
27     mu_z = X.dot(theta)
28
29     # Draw latent variable Z from its full conditional
30
31     z[y == 0] = stats.truncnorm.rvs(a = -np.inf, b = -mu_z[y == 0], loc = ↩
           mu_z[y == 0], scale=1)
32
33     z[y == 1] = stats.truncnorm.rvs(a = -mu_z[y == 1], b = np.inf, loc=↩
```

```
        mu_z[y == 1], scale=1)

    # Compute posterior mean of theta

    M = V.dot(prec_0.dot(theta_0) + X.T.dot(z))

    # Draw variable theta from its full conditional

    theta = np.random.multivariate_normal(M.ravel(),V).reshape((d,1))

    theta_chain[t,:] = theta.T

# Get the posterior mean of theta

post_theta = theta_chain[burn_in:,].mean(axis=0)
```

Outcomes of the scripts

We ran our algorithms under a standardized (random and fictitious) setting. Here follows the code of the data generating process we used at the beginning of each script:

Listing 5: Python code – Common Initial Setting

```

np.random.seed(250)

###   ###   ###   ###   ###   ###

# SET UP THE MODEL

# Set the dimension of your problem, with N as the number of observations and d as the number of parameters

n = 400
d = 4

# Generate a sequence of random numbers from a Normal distribution to then be able to fill in the matrix X

x = np.random.randn(n,d-1)

X = np.ones((n,d))
X[:,1:] = x

```

```
# Make a guess on the true value of theta

true_theta = np.array([0.5,0.2,0.3,-0.2]).reshape((d,1))

# By making use of the probit link, compute the probabilities p (that is, the mean) of the Bernoulli Multivariate model

p = stats.norm.cdf(X.dot(true_theta))

# Then again, simulate the outcomes by making use of the probabilities we have. We used here a random draw from a binomial with parameter n = 1, to get a random draw from a Bernoulli distribution

y = np.random.binomial(1, p, size=None)
```

As far as the outcome is concerned, we can focus on this specific setting and provide the means of the simulated chains after the burn-in. We decided to run all of the algorithms with 10,000 iterations and a burn-in of 5,000, to be sure that the parameters would have converged properly. In the trace plots included in the **Appendix** at the end of this paper, it is possible to see how much the averages of the simulations differed from the true values of the betas. The orange line above the histograms is the average of the post-burn-in observations, while the lilac line represents the true value of the parameter.

By running the algorithm provided in the Python Scripts attached, it is also possible to get a printed outcome with the true values of the parameters compared to the averages of the simulated data. The code in Listing 6 is meant to show the outcomes of the Informative Metropolis Algorithm (which gives the same as the Non-Informative one) and of the Auxiliary Gibbs in both its non-informative and informative specifications.

Listing 6: True parameters and final parameters obtained as output

```
# Non-Informative Metropolis Algorithm
true_theta = [ 0.5  0.2  0.3 -0.2]
final_theta = [ 0.49619663  0.27858281  0.27945457 -0.2750591 ]

# Non-Informative Auxiliary Gibbs Sampler Algorithm
true_theta = [ 0.5  0.2  0.3 -0.2]
final_theta = [ 0.49024928  0.28023793  0.27600595 -0.28028319]

# Informative Auxiliary Gibbs Sampler Algorithm
true_theta = [ 0.5  0.2  0.3 -0.2]
final_theta = [ 0.48996382  0.2800656   0.27581181 -0.28008494]
```

Basic Diagnostics

We ran basic diagnostics of the convergence of the two algorithms, taking into consideration the cases of non-informative and informative prior for both the Metropolis Algorithm and the Auxiliary Variable Gibbs Sampler Algorithm.

Since plotting the outcomes of different choices of the parameters would have implied the insertion of an exaggerate amount of graphs, we decided to run the algorithms just on these parameters:

Listing 7: Parameters chosen to check the convergence of the four algorithms

```
parameters = [0.5,0.2,0.3,-0.2]
```

The trace plots of the convergence of the four algorithms are the ones inserted right below this paragraph. We decided to follow the approach presented in class, and plotted the trace plots of the first 1000 observations after the burn-in. This way, it was possible to have a clearer picture of the convergence path of the simulated coefficients for each algorithm. From the plots it is clear how the Gibbs Sampler manages to reach a better degree of convergence compared to the Metropolis-Hastings even with 1000 iterations.

We ran all the four algorithms under similar conditions. We drew 400 observations for three parameters (leaving 400 "ones" for the constant) and looped it over 10,000 iterations with a burn-in of 5,000.

Figure 1: Convergence of Metropolis Algorithm with Non-Informative Prior (1000 iterations post burn-in)

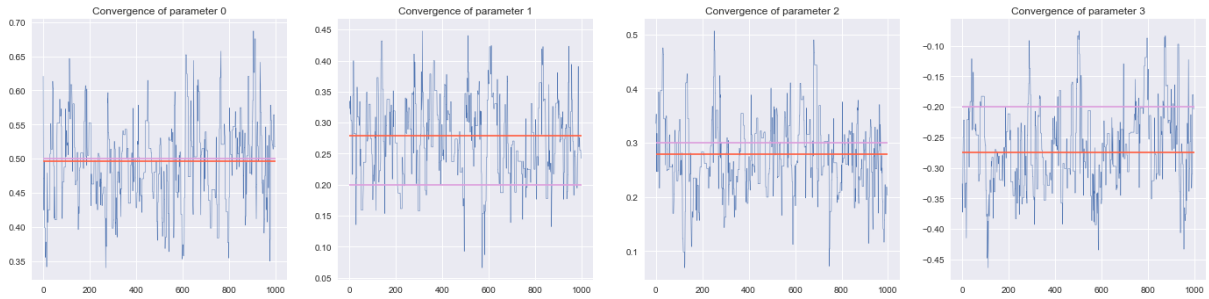


Figure 2: Convergence of Metropolis Algorithm with Informative Prior (1000 iterations post burn-in)

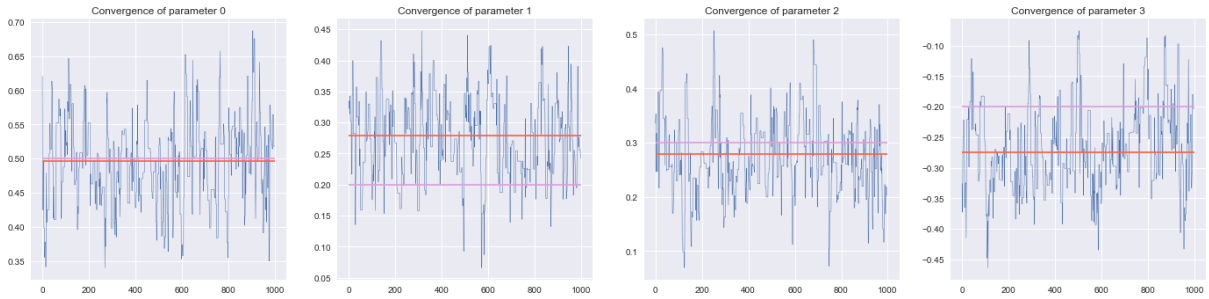
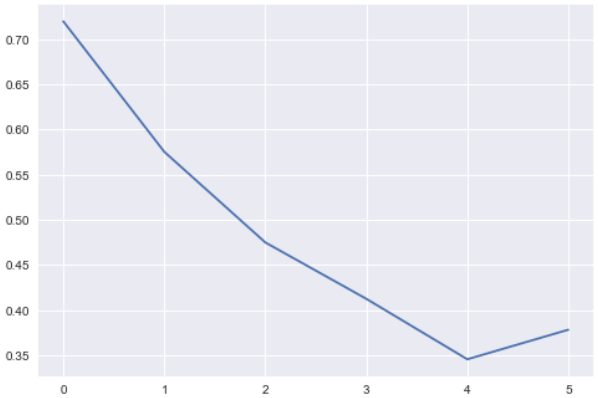


Figure 3: Convergence of Auxiliary Gibbs Sampler with Non-Informative Prior (1000 iterations post burn-in)

Figure 4: Convergence of Auxiliary Gibbs Sampler with Informative Prior (1000 iterations post burn-in)

As far as the acceptance rate is concerned, it is important to note that the Metropolis Algorithm and the Auxiliary Gibbs Sampler work differently. While the first one has an acceptance rate which is less than or equal to one (allowing the stochastic process of each parameter to stay in the same position, if the candidate is rejected), the Auxiliary Variable Gibbs Sampler is designed to accept with probability one each new candidate drawn from the Normal proposal. This being said, we successfully stored all the acceptance rates of the runs we performed for the Metropolis Algorithms with different numbers of parameters (from no parameters other than the constant, to 5 parameters other than the constant). Here are the results:

Figure 5: Acceptance rates of the Metropolis Algorithm



We decided to plot the acceptance rate for the Metropolis Algorithm with non-informative prior since they are the same as the ones of the Metropolis Algorithm with informative prior for the reasons we mentioned earlier in the report. As far as the graph is concerned, we can see

that the acceptance rate seems to converge towards the theoretical optimum level of 23 percent as we increase the number of parameters. The higher acceptance rate at smaller number of parameters also explains the bad convergence with two parameters rather than, for example, five parameters. More specifically, we should see better performance in terms of acceptance rate once the number of parameters is big enough.

Comparison of the performance of the two algorithms

To compare the performance of the two algorithms, we simulated 400 draws for each of the X's, using the numpy function drawing numbers at random from a standard Normal distribution. We then used the Probit link to simulate the dependent variables. We then ran our algorithms on this simulated dataset, by making use of the four algorithms we implemented, and stored the time it took for each algorithm to run for fixed number of parameters (from 1 to 6). Here is the sample code, where FullMethods is the Python script that includes all the four algorithms we coded.

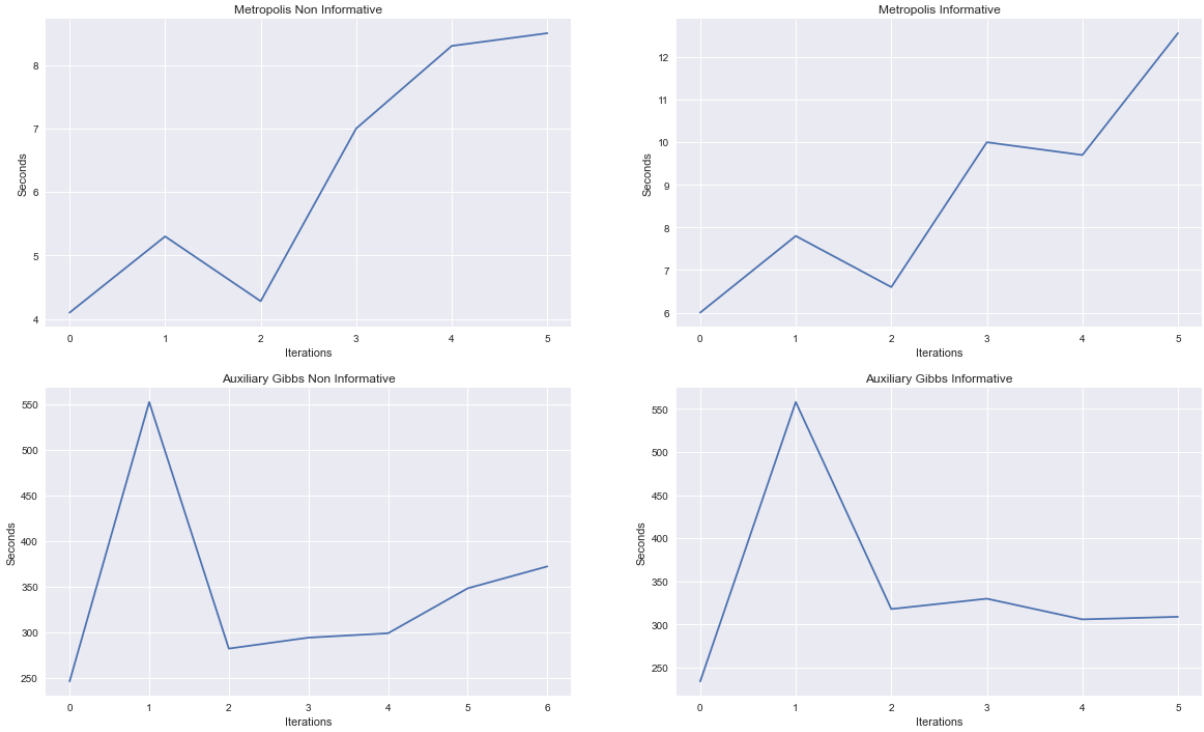
Listing 8: Python code – Storing the timing of the four algorithms and the acceptance rates

```
1
2  import FullMethods as fm
3
4  para = [[0.5],
5          [0.5,0.2],
6          [0.5,0.2,0.3],
7          [0.5,0.2,0.3,-0.2],
8          [0.5,0.2,0.3,-0.2,0.4],
9          [0.5,0.2,0.3,-0.2,0.4,0.8]]
10
11 acc_rate_NI = []
12 acc_rate_I = []
13 time_MA_NI = []
14 time_MA_I = []
15 time_AGS_NI = []
16 time_AGS_I = []
17
18 for i in range(6):
19     t1,a1 = fm.MA_NI(para[i])
20     t2,a2 = fm.MA_I(para[i])
21     t3 = fm.AGS_NI(para[i])
22     t4 = fm.AGS_I(para[i])
23     acc_rate_NI.append(a1)
24     acc_rate_I.append(a2)
25     time_MA_NI.append(t1)
26     time_MA_I.append(t2)
```

```
27      time_AGS_NI.append(t3)
28      time_AGS_I.append(t4)
```

The data stored inside the lists we created from line 13 to line 16 were then used to provide this graphical representation of the efficiency of the four algorithms:

Figure 6: Comparison of efficiency of the four algorithms



While we know that the Auxiliary Variable Gibbs Sampler should work better in a multidimensional setting, it is important to remember that we are measuring efficiency through the computation of the time it takes to each algorithm to run and provide an outcome. Letting the statistical perspective aside, and taking into consideration the computational efficiency of the algorithms we wrote, it is obvious that the Auxiliary Variable Gibbs Sampler is slower than the Metropolis Algorithm. In fact, while the Metropolis Algorithm is quite "smooth" in terms of computation, the AVGS Algorithm requires the variable Z (the auxiliary variable) to be updated each time by performing 400 draws from a truncated Normal at each iteration. As we can see from the graphs, the MH never takes more than thirteen seconds while the AVGS runtime is in the order of hundreds of seconds.

Statistically speaking, from the trace plots we provided in the **Basic Diagnostics section**, that the Auxiliary Variable Gibbs Sampler algorithm performs better than the Metropolis Algorithm. In fact, the first 1000 draws post burn-in stabilize more strongly and faster in the AVGS setting.

Derivation of the full conditionals of the Gibbs Sampler

In order to derive the full conditionals distributions used let us first remind how our model is specified. Keeping the same notation as in *Albert and Chib, 1993* the auxiliary Gibb sampler introduces N latent variables $Z_1, ..., Z_N$, where the Z_i are independent and normally distributed as $N(x_i^T\beta, 1)$ such that $Y_i = 1$ if $Z_i > 0$ and $Y_i = 0$ otherwise. From here it can be shown that the Y_i distributed as independent Bernoulli random variablew with $p_i = P(Y_i = 1) = \Phi(x_i^T\beta)$.
Now, let us first calculate the joint posterior density of the random variables β and Z, namely $\pi(\beta, Z \mid y)$. By the Bayes theorem we have:

$$\pi(\beta, Z|y) = \frac{\pi(\beta,Z)*\pi(y|\beta,Z)}{\pi(y)} \propto \pi(\beta, Z) * \pi(y|\beta, Z) = \pi(\beta) * \pi(Z|\ \beta) * \pi(y|Z)$$

since $\pi(\beta, Z) = \pi(\beta) * \pi(Z|\ \beta)$ and $\pi(y|\beta, Z) = \pi(y|Z)$ (because y and β are independent conditionally on Z).
We can rewrite the last equation keeping the prior of $\pi(\beta)$ in the same form but rewriting the likelihood $\pi(Z|\ \beta) * \pi(y|Z)$ in the following way:

$$\pi(\beta, Z|y) = \pi(\beta) * \pi(Z|\ \beta) * \pi(y|Z) = \pi(\beta) * \prod_{i=1}^{n} \pi(Z_i \mid \beta) * \pi(y_i|Z_i)$$

where

$$\pi(Z_i|\beta) = N(x_i^T\beta, 1)$$

$$\pi(y_i|Z_i) = \mathbf{1}(y_i = 1)\mathbf{1}(Z_i > 0) + \mathbf{1}(y_i = 0)\mathbf{1}(Z_i \leq 0)$$

with $\mathbf{1}(X \in A)$ is the indicator function equal to 1 if the random variable X is contained in the set A, and 0 otherwise.

At this point we acknowledge that the joint posterior is difficult to normalize and sample from directly. Luckily, computing the marginal posterior distribution of β using the Gibbs sampling algorithm requires only the posterior distribution of β conditional on Z and the posterior distribution of Z conditional on β, which are of standard forms and we can derive as follows.
Remembering once again that y and β are independent conditionally on Z and applying Bayes theorem, the full conditional of β turns out to be

$$\pi(\beta \mid Z, y) = \pi(\beta|Z) \propto \pi(\beta) * \pi(Z \mid \beta) = \pi(\beta) * \prod_{i=1}^{n} N(x_i^T\beta, 1)$$

The full conditional of Z is $\pi(Z|\beta, y)$. We already know that $\pi(Z_i|\beta) = N(x_i^T\beta, 1)$. If we condition also on y_i we need to account for the fact that the relative Z_i is either positive or negative depending on whether y_i takes value 1 or 0. Thus, the full conditional of each Z_i is given by $\pi(Z_i \mid \beta, y) = N(x_i^T\beta, 1)$ truncated ad left by 0 if $y_i = 1$ or truncated ad right by 0 if $y_i = 0$.

To conclude our discussion we note that the full conditional of β depends on the prior $\pi(\beta)$. In our algorithms we implemented two different priors and $\pi(\beta|Z, y)$ changes accordingly. If we use a non informative prior for β, namely a constant prior of form $\pi(\theta)\propto$1, the expected value of β coincides with the MLE and the posterior distribution then reduces to the Normal $N((X'X)^{-1}X'Z, (X'X)^{-1})$. In the second case, we used the proper conjugate prior, i.e. an informative normal prior $N(\beta_0, V_0)$. Here $\pi(\beta|Z, y)$ is again a Normal with $E(\beta|Z, y) = (V_0^{-1} +$

$X'X)^{-1}(V_0^{-1}\beta_0 + X'Z)$ and $cov(\beta|Z, y) = (V_0^{-1} + X'X)^{-1}$. Intuitively, the difference is that if we decide not to incorporate previous information, our estimate converges to the MLE, the same result we get in the frequentist approach, while if we use an informative prior we add to the estimation process some more information on the distribution of β in addition to the data.

Appendix: Figures

In these plots, the lilac line represents the true value of the parameter, while the orange line represents the average of the simulations.

Figure 7: Convergence of Metropolis Algorithm with Non-Informative Prior (10000 iterations)

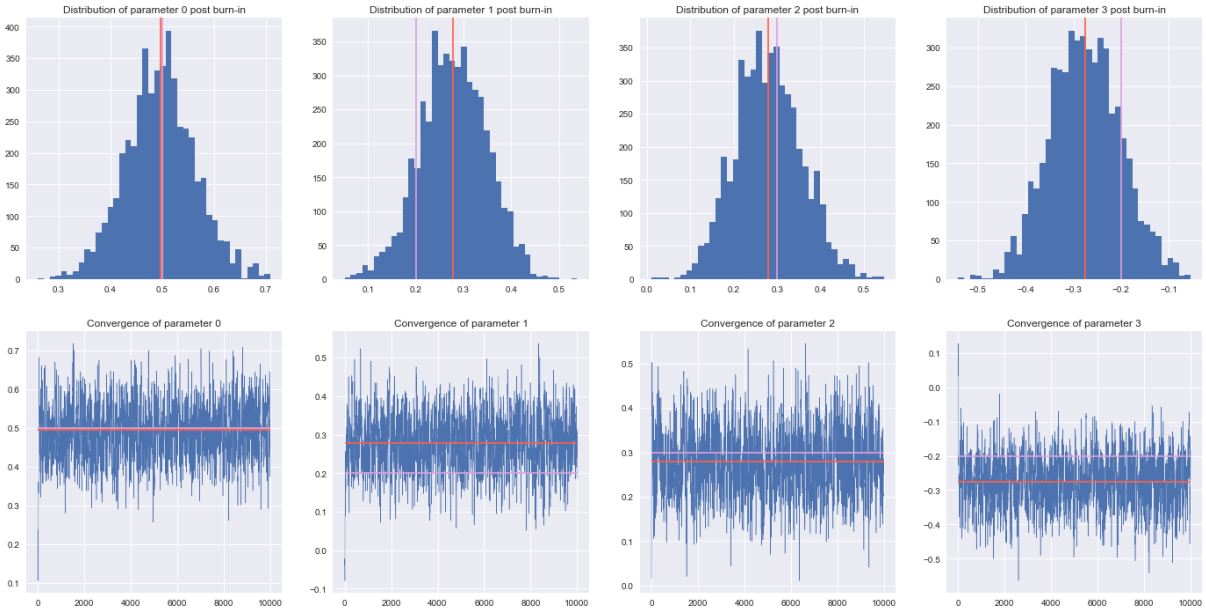


Figure 8: Convergence of Metropolis Algorithm with Informative Prior

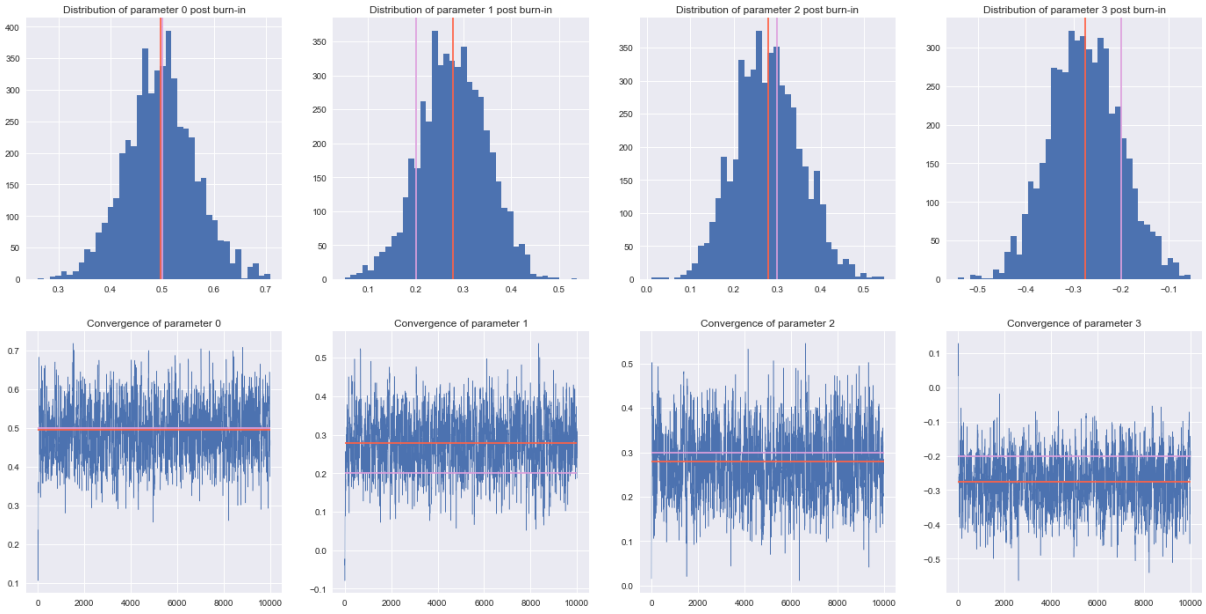


Figure 9: Convergence of Auxiliary Gibbs Sampler with Non-Informative Prior

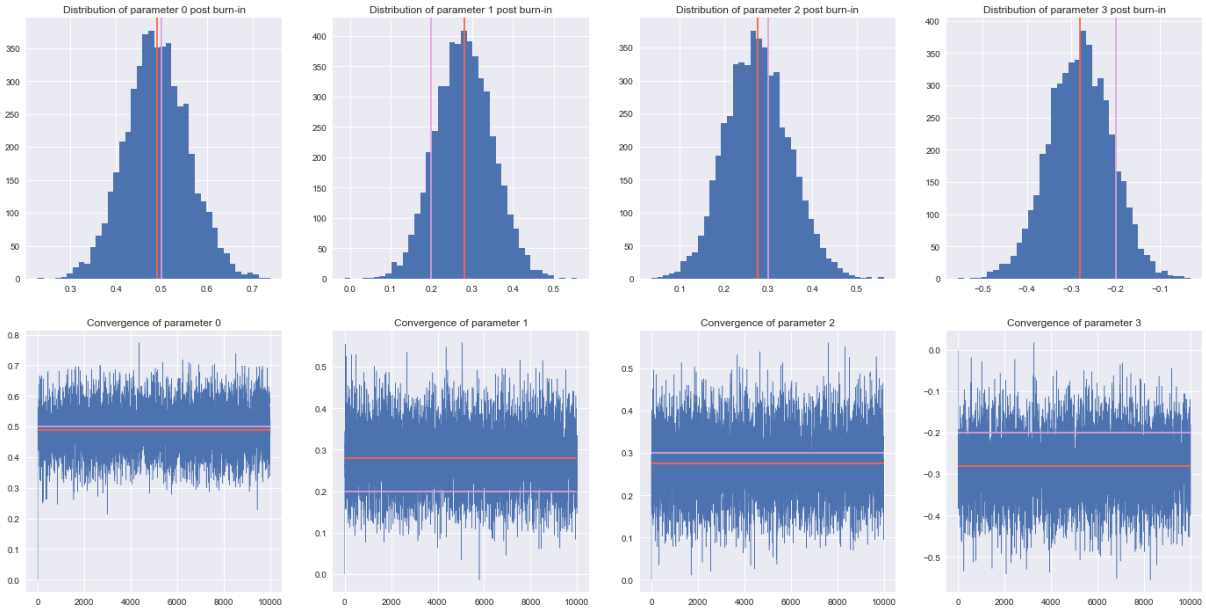


Figure 10: Convergence of Auxiliary Gibbs Sampler with Informative Prior